Lab 3 – Searching & sorting

Instructors: Lorenzo De Carli & Maan Khedr

Slides by Lorenzo De Carli

What we are going to focus on today

- Understanding the nuances of sorting algorithms
- Practicing our understanding of searching algorithms

A note on recursive algorithms

- Quicksort, mergesort, and anything using recursion may trigger the call stack limit in the Python interpreter (too many nested calls!)
- To avoid running into this issue, put this code at the beginning of your program:

```
import sys sys.setrecursionlimit(20000)
```

- Feel free to use any sufficiently large number, within reason, as a parameter. Find what works for your case.

Exercise #1 – Merge [1 pt]

- In class, we have seen an implementation of mergesort which leaves out the most important thing – the merge() function

1. Implement a basic version of that function, using the code seen in class for the rest of the algorithm [0.2 pts]
2. Argue that the overall algorithm has a worst-case complexity of $O(nlogn)$. Note, your description must **specifically refer to the code you wrote**, i.e., not just generically talk about mergesort. [0.4 pts]

Exercise #1 /2

3. Manually apply your algorithm to the input below, showing each step (similar to the example seen in class) until the algorithm completes and the vector is fully sorted. Explanation should include both visuals (vector at each step) and discussion [0.4 pts]

8	42	25	3	3	2	27	3

4. Is the number of steps consistent with your complexity analysis? Justify your answer. [0.2 pts]

What to deliver

- Submit a file named ex1.py with your implementation (question 1)
- Submit a file named ex1.pdf with your answers to questions 2, 3 and 4 (you can use any editor of your choice – Word, Google Docs, Overleaf/LaTex, etc. as long as the output is PDF)

Exercise #2 – Performance [1 pt]

- Suppose you are tasked to implement the sorting function for a algorithm library in Python
- You are told that your implementation should use bubble sort for small arrays, and quicksort for everything else
- However, they forgot to tell you what ”small” means – you’ll have to figure it out on your own

Exercise #2 /2

1. Implement and test both algorithms on input of 20 different sizes. The choice of sizes is yours, but it must be such that it evidences for which sizes each algorithm is faster [0.2 pts]
2. Each test must be repeated on best, worst, and average case scenario for both algorithm [0.4 pts]
    1. Note that “best/worst/average” may mean different things for the two different algorithms
3. Generate performance plots for all the six (three?) cases. Performance of both algorithms must be visualized in each plot. In the plots, highlight for which inputs one algorithm perform better than the other [0.4 pts]
4. Choose a threshold (on the input size) that determines whether the input is “small” or not. Justify your choice based on the plots. [0.2 pts]

What to deliver

- Submit a file named ex2.py with your implementation of both algorithms and your timing code
- Submit a file named ex2.pdf with plots (answer to question 3) and discussion (answer to question 4)
- You can use any editor of your choice – Word, Google Docs, Overleaf/LaTex, etc. as long as the output is PDF

Exercise #3 – Complexity analysis

- In class we discussed the complexity of various sorting algorithms, but we did not really get into the details of how that complexity was calculated

1. Derive the formulas for (i) number of comparisons, and (ii) average-case number of swaps for bubble sort [0.4 pts]
2. Implement a version of bubble sort that counts the number of comparisons and swaps for each execution [0.2 pts]
3. Run your code on a number of inputs of increasing size (note: inputs must be appropriate for average-case complexity analysis) [0.2 pts]
4. Separately plot the results of #comparisons and #swaps by input size, together with appropriate interpolating functions. Discuss your results: do they match your complexity analysis? [0.2 pts]

What to deliver

- Submit a file named ex3.py with your implementation of bubble sort, instrumented as discussed in question 2. The file should also implement the measurements discussed in question 3.
- Submit a file named ex3.pdf with answers to questions 1 and 4
- You can use any editor of your choice – Word, Google Docs, Overleaf/LaTex, etc. as long as the output is PDF

Exercise #4 – More complexity analysis [1 pt]

- In class we have discussed how quicksort's worst case complexity is $O(n^2)$, but its average-case complexity is $O(nlogn)$

1. Derive the formula for worst-case complexity [0.4 pts]
2. Come up with a vector of 16 elements which incurs worst-case complexity. Manually show the workings of the algorithm until the vector is sorted. [0.2 pts]
3. Run an implementation of quicksort on a number of inputs of increasing size which incur worst-case complexity [0.2 pts]
4. Plot the results, together with appropriate interpolating functions and discuss your results: do they match your complexity analysis? [0.2 pts]

What to deliver

- Submit a file named ex4.py with your implementation of quick sort, and the measurement code discussed in question 3.
- Submit a file named ex4.pdf with answers to questions 1, 3 and 4
- You can use any editor of your choice – Word, Google Docs, Overleaf/LaTex, etc. as long as the output is PDF

Exercise #5 – Sorting for real [1 pt]

- We discussed that Python uses an algorithm different than the ones we have seen in class. It is called Timsort, and it is pretty complicated
  - Internally, it uses an algorithm called binary sort (or binary insertion sort) to sort small sub-arrays; this algorithm is an optimization of insertion sort
- First, learn about binary insertion sort:
  - https://www.geeksforgeeks.org/binary-insertion-sort/
  - https://www.interviewkickstart.com/learn/binary-insertion-sort

Exercise #5 /2

1. Implement both “traditional” insertion sort and binary insertion sort [0.3 pts]
2. Devise and run an experiment where you test each algorithm on a number of average-case inputs of increasing length [0.3 pts]
3. Plot the results of both algorithms within the same plot, together with appropriate interpolating functions [0.2 pts]
4. Discuss the results: which algorithm is faster? Why? [0.2 pts]

What to deliver

- Submit a file named ex5.py with your implementation of the algorithms (question 1), measurement code (question 2), and plotting code (question 3)
- The code must also contain the answer to question 4 as code comments

Exercise #6 –Sorting + Searching [1 pt]

- Binary search is faster than linear search, but requires the data to be sorted
- How much of a problem is this?
- Consider two search algorithms:
  - The first is just linear search
  - The second uses quicksort to sort the input, then searches it using binary search
  - Which one is faster? When? This exercise will look into it

Exercise #6 /2

1. First, implement both algorithms [0.1 pts]
2. Then, measure their performance on 100 random tasks. An appropriate random task here would be to search for a constant element in an array that gets reshuffled every time [0.1 pts]
   - Refer to the code for lecture 7 for inspiration
3. You must redo the above with inputs of size 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000 [0.2 pts]
4. Plot the above and discuss which algorithm is faster [0.1 pts]
5. Redo the above with an input that causes quicksort to incur worst-case performance [0.5 pts]

What to deliver

- Submit a file named ex6_avg.py, implementing code, measurement and plots from questions 1-4
- The code should also answer the discussion part of question 4 as code comments
- Submit a file named ex6_worst.py, implementing all of the above for question 5

Exercise #7 – More interpolation search [1 pt]

- Interpolation search is a variant of binary search, in which search is started for a location other than an array's midpoint
- How does the choice of midpoint affect performance? Let's find out!
- You are given an array at: https://github.com/ldkteaches-calgary/ensf338W24/blob/main/lab03/ex7data.json
- You are also given a set of tasks at: https://github.com/ldkteaches-calgary/ensf338W24/blob/main/lab03/ex7tasks.json
- The list of search tasks is just a list of numbers; the goal of the task is to determine whether each input is in the array

Exercise #7 /2

1. Implement a standard binary search, with the following tweak: the midpoint for the first iteration must be configurable (all successive iterations will just split the array in the middle) [0.2 pts]
2. Time the performance of each search task w/ different midpoints for each task. You can use whatever strategy you want to check different midpoints. Then, choose the best midpoint for each task [0.4 pts]
3. Produce a scatterplot visualizing each task and the corresponding chosen midpoint [0.2 pts]
4. Comment on the graph. Does the choice of initial midpoint appear to affect performance? Why do you think is that? [0.2 pts]

What to deliver

- Submit a file named ex5.py with your implementation of the algorithms (question 1), measurement code (question 2), and plotting code (question 3)
- The code must also contain the answer to question 4 as code comments

How to submit

- Upload a zip file to the “Lab 3” dropbox on D2L, containing the required content for every exercise

Grading rubric

- You get **3 pts** for uploading a **partial solution** by **end of lab**
  - **Must not be an empty file or irrelevant material**
- Then, you'll have until **11:59PM of the day before the next lab** to upload the **complete solution**. That will be graded as follows:
  - Exercise 1-7: 1 pts each
  - **Can upload the complete solution to the same dropbox**

That’s all folks!